Managing Inheritance Hierarchies in Object/Relational Mapping Tools

Luca Cabibbo and Antonio Carosi

Dipartimento di Informatica e Automazione
Università degli studi Roma Tre

Abstract. We study, in the context of object/relational mapping tools, the problem of describing mappings between inheritance hierarchies and relational schemas. To this end, we introduce a novel mapping model, called M^2ORM^2+*HIE*, and investigate its mapping capabilities. We first show that M^2ORM^2+*HIE* subsumes three well-know basic representation strategies for mapping a hierarchy to relations. We then show that M^2ORM^2+*HIE* also allows expressing further mappings, e.g., where the three basic strategies are applied independently to different parts of a multi-level hierarchy. We describe the semantics of M^2ORM^2+*HIE* in term of how CRUD (i.e., Create, Read, Update, and Delete) operations on objects (in a hierarchy) can be translated into operations over a corresponding relational database. We also investigate correctness conditions.

1 Introduction

Enterprise applications are often developed using an object-oriented programming language (e.g., Java or C#) and a relational database. In this common case, applications need to load data from the database, create objects to represent this data in main memory, perform computations involving these objects, and store changes to objects back in the database. *Object/relational mapping* tools (or, simply, *ORM* tools) are frameworks for storing and retrieving persistent objects; their goal is to support the complex activity of managing the connections between objects and a relational database. ORM tools allow the programmer to manage the persistence of objects by means of standard API's, such as the JDO [11] or the ODMG ones [7], that is, the same way he would use objects in an object database. Persistence is transparent to the programmer, since he does not know actual implementation details. The bridge between objects and underlying relations is realized on the basis of a data mapping specification.

The *meet-in-the-middle* approach is a mapping strategy for ORM tools. It assumes that data classes (e.g., classes in the application logic holding persistent data) and a relational database have been developed in an independent way. The developer should also describe the correspondences between data classes and the relational database. These correspondences describe a "meet in the middle" between the object schema and the relational schema; they are used by the ORM tool to let objects persist by means of the database. The meet-in-the-middle

approach is very versatile, since modifications in data classes and/or in the relational database can be managed by simply redefining the correspondences. There are several object/relational mapping tools offering the meet-in-the-middle approach (e.g., [10, 13, 15, 16]). However, current systems support the meet-in-the-middle approach still in a limited way, especially because they allow defining only rather restricted kinds of correspondences.

A main limitation imposed by current ORM tools concerns the representation of inheritance hierarchies. There are three well known main strategies to represent a class with its subclasses in a relational schema [2, 9]: (i) by using a single relation; (ii) a relation for each class; and (iii) a relation for each concrete (sub-)class (especially if the superclass is abstract). in practice, current ORM tools do not always offer all the three basic representation strategies for inheritance hierarchies. Furthermore, they usually permit to select, for each hierarchy, a single representation strategy, to be applied to the whole hierarchy.

In previous work [5, 6], we have introduced M^2ORM^2 (*Meet-in-the-Middle Object/Relational Mapping Model*), a model to describe mappings between object schemas and relational schemas, to support the transparent management of object persistence based on the meet-in-the-middle approach. In M^2ORM^2 it is possible to express complex correspondences between groups of related classes and groups of related relations; it is also possible to express correspondences describing relationships between groups. It turns out that M^2ORM^2 generalizes and extends the kinds of correspondences managed by current proposals and systems, e.g., [10, 11, 13, 15, 16], thus allowing for more possibilities to meet schemas. However, until now, we did not take account of inheritance hierarchies in M^2ORM^2.

In this paper, we study, in the context provided by M^2ORM^2, the problem of establishing mappings between inheritance hierarchies and relational schemas. To this end, we introduce a novel mapping model, called M^2ORM^2+*HIE*, and investigate its mapping capabilities. We show that M^2ORM^2+*HIE* subsumes the three above cited basic representation strategies for mapping hierarchies to relations. Moreover, we show that it also allows expressing further mappings, e.g., where the three basic strategies are applied independently to different parts of a multi-level hierarchy. We present the structure and semantics of M^2ORM^2+*HIE* mappings; more specifically, we describe how CRUD (i.e., Create, Read, Update, and Delete) operations on objects in an inheritance hierarchy can be translated into operations over a corresponding relational database. We also discuss the problem of verifying the correctness of M^2ORM^2+*HIE* mappings involving hierarchies. Again, it turns out that M^2ORM^2+*HIE* generalizes and extends the kinds of correspondences managed by current systems.

The paper is organized as follows. Section 2 recalls some preliminary notions, including the M^2ORM^2 mapping model and the three basic representation strategies for inheritance hierarchies. Section 3 introduces M^2ORM^2+*HIE*, together with a number of examples. The semantics of M^2ORM^2+*HIE* is described in Section 4. Section 5 discusses conditions for the correctness of M^2ORM^2+*HIE* mappings. Finally, Section 6 discusses related work and Section 7 draws some conclusions.

2 Preliminaries

In this section we briefly present the data models and the terminology used in this paper. We also describe basic notions concerning object/relational mappings based on the meet-in-the-middle approach. Finally, we recall three basic representation strategies for representing hierarchies by means of relations.

2.1 Data models

Object model. We consider a simple but realistic *object model*, similar to that of UML [4] and ODMG [7]. We focus on the structural features of the model.

A *class* is a set of *objects* having the same structural (and behavioral) properties. Each class has a set of *attributes* associated with it. In this paper we make the simplifying hypothesis that class attributes are of a same simple type, e.g., strings. A *class with key* is a class in which each object can be identified on the basis of the value of one of its attributes, called the *key attribute* of the class.

An *association* is a binary relation between a pair of classes, whose instances are *links* between pairs of objects belonging to the two classes. We consider navigability and multiplicity of (roles of) associations.

Classes are also related by *generalization/specialization relationships*. This relationship is also called *inheritance* in object-oriented programming, since it implies attribute, association, and method inheritance from the superclass to the subclass. We consider single inheritance only. The inheritance relationship induces *inheritance hierarchies* (or, simply, *hierarchies*) on classes. A hierarchy is a (maximal) rooted tree of classes connected by inheritance relationships. Because of inheritance, an object may belong to multiple classes; indeed, if an object belongs to a class C, it belongs to all the superclasses of C as well. However, as it is customary in object-oriented programming, we assume that each object o has a unique *most specific class*, that is, a class C such that o belongs *only* to C and the superclasses of C. In a hierarchy, a class C is *abstract* if every object that belongs to C must also belong to some subclass of C, that is, if there cannot be objects whose most specific class is C. By contrast, a class that is not abstract is called *concrete*. In hierarchies, leaf classes should be concrete.

An *object schema* is a set of classes, together with associations and inheritance relationships among such classes. Figure 1 shows a sample object schema, comprising a hierarchy. Constraint $\{key\}$ denotes key attributes.

Relational model. In the *relational model* [8], a *relation schema* is a set of *attributes*. We assume that all relation attributes are of a simple type, e.g., strings. A *relational schema* is a set of relation schemas. At the instance level, a *relation* is a set of *tuples* over the attributes of the relation.

We consider the following integrity constraints. Attributes can be or not be *non null*. Each relation has a *primary key* (or, simply, *key*). A *key attribute* is an attribute that belongs to a key; key attributes must be non null. A *foreign key* is a non-empty set of attributes of a relation used to reference tuples of another relation; foreign keys define *referential constraints* between relations.

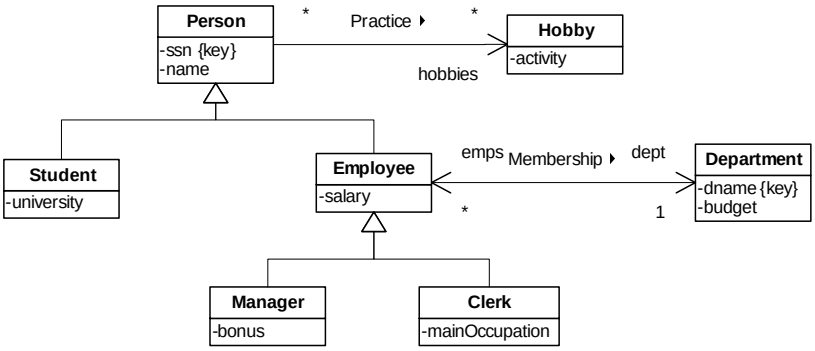


Fig. 1. An object schema

Figure 2 shows a sample relational schema. Attributes forming primary keys are underlined. Referential constraints are denoted by arrows.

2.2 Object/relational mappings

Object/relational mapping tools. When an ORM tool is used, objects and links are manipulated by means of *CRUD* operations (*Create*, *Read*, *Update*, *Delete*), which allow the programmer to create persistent objects, to read persistent objects (that is, perform a unique search of an object based on its key), as well as to modify and delete persistent objects. Navigation, formation, breaking, and modification of persistent links between objects are also possible. In correspondence to such programmatic manipulations of an object schema, an ORM tool should translate CRUD operations on objects and links into operations over the underlying relational database. This translation should happen in an automatic way, on the basis of a suitable mapping between the object schema and the relational schema, as described next and in the following of this paper.

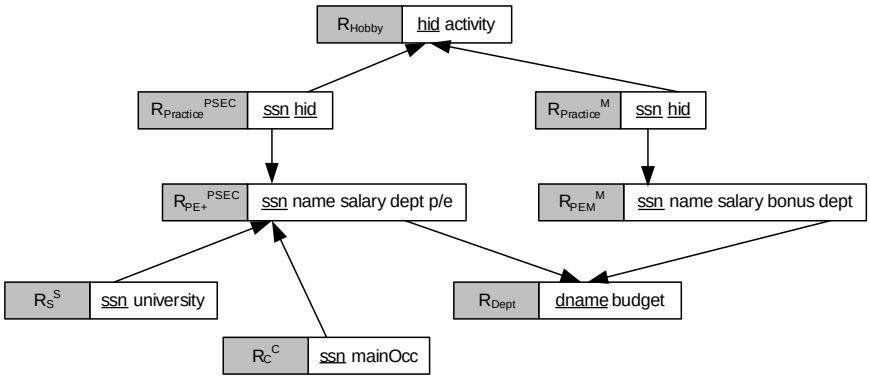


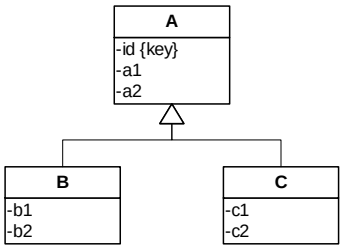
Fig. 2. A relational schema

The M^2ORM^2 mapping model. We now briefly describe the M^2ORM^2 mapping model [5, 6]. (We refer the reader to our previous work on M^2ORM^2 for a more detailed presentation of this mapping model.) For the sake of presentation, we assume here that the object schema does not contain hierarchies. (This limitation will be removed next.) In M^2ORM^2, a *mapping* between an object schema and a relational schema is represented as a graph, comprising nodes and arcs. A *node* describes the correspondences between one or more classes and one or more relations. Usually, a node contains just one class; if there are more than one, a class is selected as the *primary class* of the node and it is related to other (*secondary*) classes in the node by associations. Similarly, a node contains usually just one relation; if there are more than one, a relation is selected as the *primary relation* of the node and it is related to other (*secondary*) relations in the node by referential constraints. The goal of a node is to describe how to represent an object of the primary class (and possibly further related objects from secondary classes) by means of a tuple in the primary relation (and possibly further related tuples in secondary relations). In a node, data (values) flow between objects and tuples as described by *attribute correspondences*, each relating a class attribute to a relation attribute. A mapping can also contain *relationship arcs*, each describing the correspondences between a pair of nodes by relating an association (between the primary classes of the two nodes) and one or more referential constraints (involving the primary relations of the nodes, and possibly others). The semantics of M^2ORM^2 is described in Section 4.1.

2.3 Basic strategies for mapping inheritance hierarchies

The problem of mapping a hierarchy to a set of relations is described in several textbooks (e.g., [1, 2, 9]) where, among others, three main basic representation strategies are considered. We now describe these strategies, and exemplify their application to the simple inheritance hierarchy shown in Fig. 3. In giving names to relations used to represent hierarchies, we write R^T_S to denote the fact that this relation has attributes for classes in the set S and that it contains a tuple for each object whose most specific class is a class in the set T. We also write $C\uparrow$ to denote the set comprising a class C together with its superclasses, and $C\downarrow$ to denote the set comprising a class C together with its subclasses. Finally, symbol + denotes that a *type* attribute is used.

- **Single Relation inheritance** (**SR**): A hierarchy H is represented by a single relation R^H_{H+}. Relation R^H_{H+} has attributes for all the attributes of classes in H; furthermore, R^H_{H+} has a *type* attribute to indicate the most specific class for the object represented by a tuple. An object in the hierarchy is represented by a single tuple in relation R^H_{H+}. For the hierarchy of Fig. 3, *SR* leads to a single relation $R^{ABC}_{ABC+}(\underline{id}, a1, a2, b1, b2, c1, c2, t)$, where t is the type attribute, whose possible values are A, B, or C. *SR* is called *Single Table inheritance* in [9].
- **Class Relation inheritance** (**CR**): A hierarchy H is represented by a relation $R^{C\downarrow}_C$ for each class C in H. Each relation $R^{C\downarrow}_C$ has attributes for the



SR	$R^{ABC}_{ABC+}(\underline{id}, a1, a2, b1, b2, c1, c2, t)$
CR	$R^{ABC}_{A}(\underline{id}, a1, a2)$ $R^{B}_{B}(\underline{id}, b1, b2)$ $R^{C}_{C}(\underline{id}, c1, c2)$
CCR A is abstract	$R^{B}_{AB}(\underline{id}, a1, a2, b1, b2)$ $R^{C}_{AC}(\underline{id}, a1, a2, c1, c2)$
CCR A is concrete	$R^{A}_{A}(\underline{id}, a1, a2)$ $R^{B}_{AB}(\underline{id}, a1, a2, b1, b2)$ $R^{C}_{AC}(\underline{id}, a1, a2, c1, c2)$

Fig. 3. A simple hierarchy and three basic relational representations

attributes of the class C it represents; a relation for a subclass also has a foreign key towards the relation for its (direct) superclass in the hierarchy. An object in the hierarchy is represented by multiple tuples: if the most specific class for the object is C, by a tuple in relation $R^{C\downarrow}_{C}$ which represents C, together with a tuple for each relation $R^{C'\downarrow}_{C'}$ that represents a superclass C' of C. For the hierarchy of Fig. 3, *CR* leads to relations $R^{ABC}_{A}(\underline{id}, a1, a2)$, $R^{B}_{B}(\underline{id}, b1, b2)$, and $R^{C}_{C}(\underline{id}, c1, c2)$, where attribute *id* in relations R^{B}_{B} and R^{C}_{C} references relation R^{ABC}_{A}. *CR* is called *Class Table inheritance* in [9].

– **Concrete Class Relation inheritance** (**CCR**): A hierarchy H is represented by a relation $R^{C}_{C\uparrow}$ for each concrete class C in H. Each relation $R^{C}_{C\uparrow}$ has attributes for the attributes of the concrete class C it represents, but also for attributes for each superclass of C. An object in the hierarchy is represented by a single tuple in the relation $R^{C}_{C\uparrow}$ for the most specific class C of the object. For the hierarchy of Fig. 3, assuming that class A is abstract, *CCR* leads to relations $R^{B}_{AB}(\underline{id}, a1, a2, b1, b2)$ and $R^{C}_{AC}(\underline{id}, a1, a2, c1, c2)$. However, if A is concrete, *CCR* leads to relations $R^{A}_{A}(\underline{id}, a1, a2)$, $R^{B}_{AB}(\underline{id}, a1, a2, b1, b2)$, and $R^{C}_{AC}(\underline{id}, a1, a2, c1, c2)$. *CCR* is called *Concrete Table inheritance* in [9] and *One inheritance path one table* in [12].

To the best of our knowledge, current object/relational mapping tools adopt mainly these three basic strategies, even tough other representation strategies for inheritance hierarchies are known (e.g., *Map classes to a generic structure* [1]).

The basic representation strategies can be applied to an inheritance hierarchy as a whole. In case of a multi-level hierarchy (e.g., where a subclass can have its own sub-subclasses, etc.), it is also possible to apply different strategies to distinct parts of the hierarchy. A possible advice is to apply the representation procedure recursively, starting from the bottom of the hierarchy and representing one inheritance level at a time [2].

In practice, current ORM tools do not always offer all the three above cited basic representation strategies for hierarchies. Furthermore, they usually permit to select, for each hierarchy, a single representation strategy, to be applied to the whole hierarchy. (See Section 6 for a discussion on the management of hierarchies

in current tools.) These facts limit the number of possible mappings that can be identified among an object schema with hierarchies and a relational database.

3 A model for mapping hierarchies and relations

We now describe a mapping model for dealing with inheritance hierarchies. Specifically, we extend M^2ORM^2 to represent hierarchies by means of a novel kind of arcs, called inheritance arcs. This extension is called M^2ORM^2+*HIE* (*M^2ORM^2 with inheritance HIErarchies*).

In M^2ORM^2+*HIE*, a *node* describes the correspondences between a primary class and a primary relation, possibly involving other classes (related by associations and/or inheritance) and other relations (related by referential constraints).

In nodes, apart from *attribute correspondences* (each relating a class attribute with a relation attribute), relation attributes can also be related to constant values by means of *literal correspondences*.

An *inheritance arc* from a node N_2 to a node N_1 represents an inheritance relationship from the primary class C_2 of N_2 to the primary class C_1 of N_1, that is, the fact that C_2 is a (direct) subclass of C_1. Normally, an inheritance arc specifies that attribute and literal correspondences are inherited from the node for the superclass to the node for the subclass. However, correspondences in a node can override inherited correspondences.

An inheritance arc can have an associated *foreign key correspondence*, from a key attribute of the primary relation in the node for the subclass to the key attribute of the primary relation in the node for the superclass. This specifies a referential constraint between the two relations.

In M^2ORM^2+*HIE*, some elements can be abstract. An *abstract node* contains just an abstract class, but no relations. An abstract node defines, implicitly, an *abstract attribute correspondence* for each attribute of the class; intuitively, abstract correspondences specify correspondences that should be provided by nodes where the abstract correspondences are inherited. An abstract node cannot contain "concrete" correspondences. An *abstract inheritance arc* specifies that the node for the subclass does not inherit correspondences from the node for the superclass; rather, these correspondences should be considered abstract correspondences, and they should therefore be redefined in the node for the subclass. Intuitively, an abstract inheritance arc towards the node for a concrete class is similar to an inheritance arc towards the node for an abstract class.

As in M^2ORM^2, relationship arcs are also allowed [5, 6].

The following example shows how the mappings implied by the three basic representation strategies of Section 2.3 can be described in M^2ORM^2+*HIE*.

Example 1. Consider the schemas of Fig. 3.

In M^2ORM^2+*HIE*, the mapping for the translation implied by *SR* can be represented using a node for each class, N_A, N_B, and N_C, together with inheritance arcs between them, that is, from N_B to N_A and from N_C to N_A. Each node relates a class of the hierarchy to relation R^{ABC}_{ABC+}; moreover, each node has an attribute correspondence for each attribute of the class of the node (relating it to

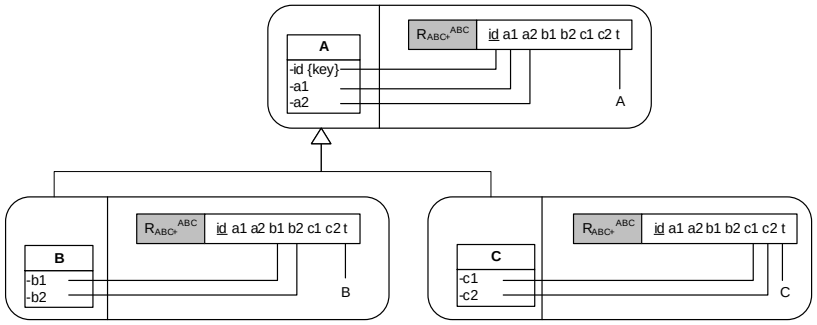


Fig. 4. *SR* in M^{2}ORM^2+*HIE*

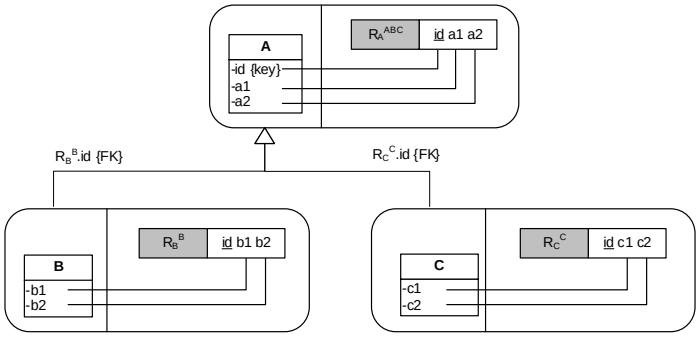


Fig. 5. *CR* in M^{2}ORM^2+*HIE*

the corresponding relation attribute, e.g., $A.a1$ to $R^{ABC}_{ABC+}.a1$). Finally, node N_A has literal correspondence $R^{ABC}_{ABC+}.t = A$, whereas nodes N_B and N_C override it as $R^{ABC}_{ABC+}.t = B$ and $R^{ABC}_{ABC+}.t = C$, respectively.[1] Figure 4 shows this mapping.

The mapping for *CR* can be represented, again, by using three nodes and two arcs, as it is shown in Fig. 5. Each node relates a class of the hierarchy to the corresponding relation, e.g., node N_A relates class A to relation R^{ABC}_A; each node has attribute correspondences for the attributes of the class of the node. In this case, inheritance arcs carry further information needed to complete the mapping specification; in particular, the inheritance arc from N_B to N_A holds a foreign key correspondence from $R^B_B.id$ to $R^{ABC}_A.id$. The case for the arc from N_C to N_A is similar.

Figure 6 shows the mapping for *CCR* when class A is abstract. Again, three nodes are needed, as well as inheritance arcs between them. Node N_A is abstract, and as such it has abstract correspondences for attributes of class A. Nodes N_B and N_C relate concrete classes B and C to relations R^B_{AB} and R^C_{AC}, respectively. Each node for a concrete class has attribute correspondences for the attributes

[1] Literal correspondences such as $R^{ABC}_{ABC+}.b1 = \mathit{null}$ in N_A are not needed, since this is the default in M^{2}ORM^2

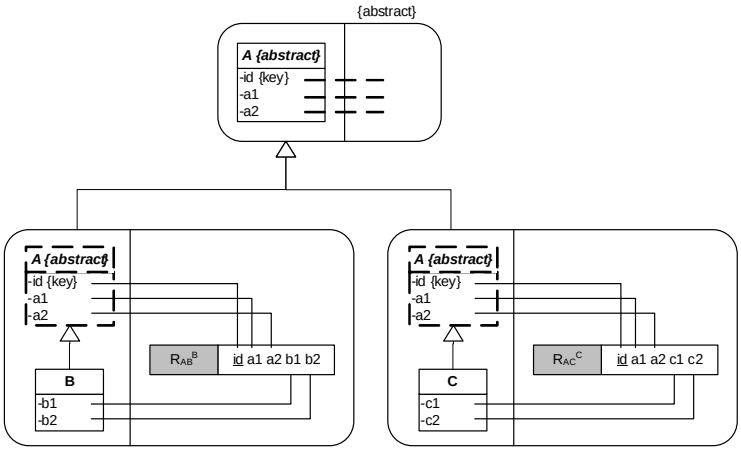


Fig. 6. *CCR* in M^{2}ORM^2+*HIE* (class A is abstract)

of the class of the node, but also for the attributes of their abstract superclass for which an abstract attribute correspondence is inherited, e.g., node N_B has an attribute correspondence between $A.a1$ and $R^B_{AB}.a1$.

The mapping for *CCR* when A is concrete is shown in Fig. 7. Node N_A relates class A to relation R^A_A. The mapping contains also abstract inheritance arcs from N_B to N_A and from N_C to N_A. Node N_B relates class B to relation R^B_{AB}. N_B has attribute correspondences for the attributes of B, but also for the attributes of its superclass A, e.g., node N_B has an attribute correspondence between $A.a1$ and $R^B_{AB}.a1$. The case for node N_C is similar. □

Besides mappings corresponding to the three basic representation strategies for hierarchies described in Section 2.3, M^{2}ORM^2+*HIE* allows specifying more complex mappings, as the following example shows.

Example 2. Consider the hierarchy in the object schema of Fig. 1. Assume all the classes are concrete. Figure 8 shows a complex mapping between this hierarchy and the relational schema of Fig. 2. (We use letters P, S, E, M, and C to denote classes *Person*, *Student*, *Employee*, *Manager*, and *Clerk*, respectively.) Relation R^{PSEC}_{PE+} has tuples for *Person*s, *Student*s, *Employee*s, and *Clerk*s (but not for *Manager*s); relation R^S_S holds further data for *Student*s, whereas relation R^C_C holds further data for *Clerk*s; finally, relation R^M_{PEM} holds all information for *Manager*s. Five nodes are used, together with the three different kinds of arcs (i.e., with and without foreign key correspondence, and abstract). We can think of this relational schema not as obtained by applying a single representation strategy to the whole hierarchy, but rather by applying different strategies to distinct parts of the same hierarchy. □

We would like to point out that, to the best of our understanding, none of the systems we have analyzed (including, among others, [10, 13, 15]) is able to manage a mapping similar to the one described by Example 2.

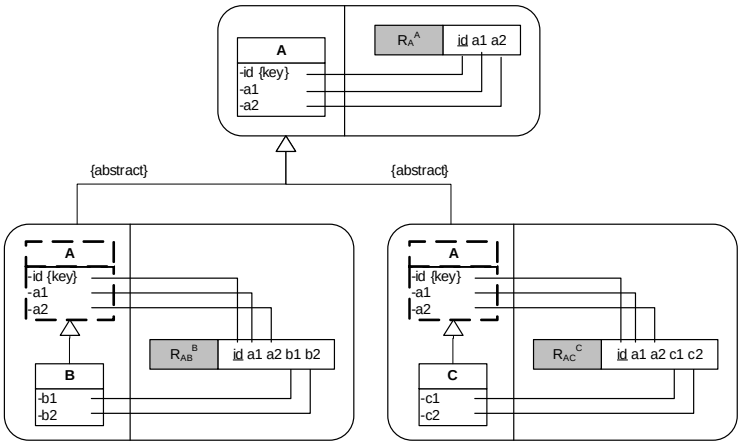


Fig. 7. *CCR* in M^2ORM^2+*HIE* (class A is concrete)

4 Semantics of mappings

In this section we present the semantics of M^2ORM^2+*HIE*. We first briefly recall the semantics of M^2ORM^2 [5, 6] (where inheritance is not allowed).

4.1 Semantics of M^2ORM^2

In M^2ORM^2, a node maps a group of classes (a primary class connected by one-to-one or many-to-one associations to further secondary classes) to a group of relations (a primary relation connected by referential constraints to further secondary relations). The goal of a node is to represent an object o of the primary class, together with objects in secondary classes that are reachable from o by means of associations represented within the node, as a tuple t_o in the primary relation, together with tuples in secondary relations that are reachable from t_o by means of referential constraints represented within the node.

In general, we have the following intuitive semantics for CRUD operations (applied to an object in the primary class of a node): The *creation* of an object o in the primary class is managed as the insertion into the database of tuples (in the primary and in secondary relations) representing both object o and objects in secondary classes that are reachable from o; values flow from object (i.e., class) attributes to tuple (i.e., relation) attributes. To *read* an object of the primary class, given its key, a query over the database is executed to retrieve the tuples (in the primary and in secondary relations) that represent an object o in the primary class and objects in secondary classes that are reachable from o; then, the corresponding objects are created in memory; values flow from tuple attributes to object attributes. The *update* of attributes of an object (or the update of links between them, thereof) is managed by modifying the tuples used to represent the group of objects. The *deletion* of an object o in the primary class

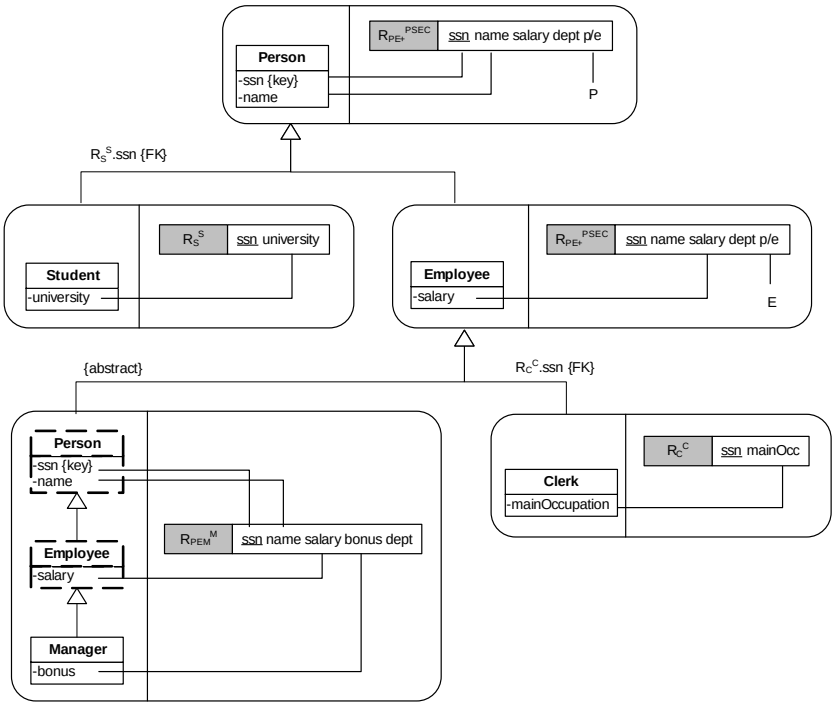


Fig. 8. A complex mapping over the hierarchy of Fig. 1

is managed by deleting the tuple t_o in the primary relation used to represent object o.

We do not describe the semantics of association arcs, since its knowledge is not needed here.

4.2 Semantics of M^{2}ORM^2+*HIE*

Before defining the semantics of M^{2}ORM^2+*HIE*, it is worth to note that, in a mapping, a relation may be used to contain tuples representing objects from multiple most specific classes. For example, in the mapping shown in Fig. 8, relation R_{PE+}^{PSEC} contains a tuple for each object whose most specific class is either *Person*, *Student*, *Employee*, or *Clerk*. We say that, in a mapping, a relation R *stores* class C if it is intended to contain, among others, a tuple for each object whose most specific class is C. We have the following characterization for the "stores" relationship:

- let C be a concrete class; the set of relations that *store* C can be computed by visiting the node whose primary class is C and all the nodes that can be reached from it by climbing up inheritance arcs that are not abstract;
- let R be a relation; the set of classes that *are stored by* R can be computed by visiting each node containing R and all the nodes that can be reached

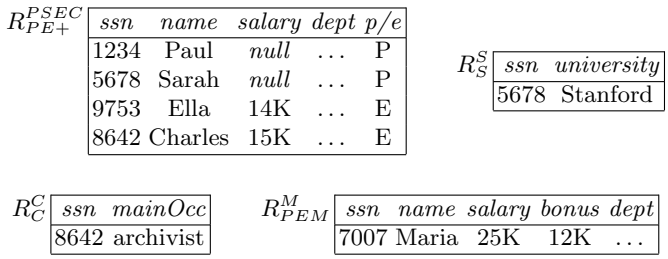


Fig. 9. Relations store classes

from them by going down inheritance arcs that are not abstract (primary classes only).

Figure 9 shows how relations store classes with respect to the mapping described in Example 2.

We can now define the semantics for CRUD operations (applied to objects in an inheritance hierarchy).

Creation. Consider the creation of an object o in the primary class C of a node. This class will be the most specific class for the object to be created. Object o has values for the attributes defined in class C, but also for attributes defined in superclasses of C. Object o will be represented by a tuple for each relation that stores C; these relations belong to a path, in the graph describing the mapping, from a node for some superclass C' of C to the node for C. Values flow from attributes of o to tuples representing o, as described by attribute and literal correspondences. Specifically, attribute and literal correspondences are applied, in sequence and downwards, from the node for C' to the node for C. Tuples that are identified in this way are inserted into the database.

For example, consider the mapping described by Example 2. The creation of an object in class *Clerk* would involve relations R^{PSEC}_{PE+} and R^{C}_{C}, which store *Clerk*. The node for *Person* specifies the value for attributes *ssn* and *name* in R^{PSEC}_{PE+}. In the same relation, the node for *Employee* specifies the value for attributes *salary* and *p/e* (the literal correspondence in this node overrides the one in the node for *Person*). Finally, the node for *Clerk* specifies the value for the tuple to be inserted in relation R^{C}_{C}.

Reading. Consider the reading of an object from a class C, given its key *id*. When performing this operation, the retrieved object o should belong to the most specific class among C and the subclasses of C (this is known as *polymorphic reading*). Therefore, the reading starts by identifying, by issuing a number of database queries $q_1, q_2, \ldots$, the most specific class C' for the object o whose key is *id*. Each query q_i is relative to some concrete class C_i that is either C or some subclass of C, and has the goal of checking whether the database represents an object belonging to C_i whose key is *id*. Query q_i comprises a join of the relations that store C_i, with selections for the key *id* and for (possibly inherited) literal

correspondences in the node for C_i. The class C' for the object we are reading is chosen as the most specific class (i.e., the most downwards in the hierarchy) among those to which the object can belong. Then, the reading proceeds as in the standard semantics of M^2ORM^2, by performing a query over the relations that store C' and by creating, in memory, the desired object o. Values flow from attributes of the retrieved tuples to o (and possibly other related objects), as described by attribute correspondences.

Consider again the mapping of Example 2. Assume that a (polymorphic) reading over class *Employee* has been requested, given the ssn *id*. Three queries q_e, q_m, and q_c are performed, to check whether an *Employee*, a *Manager*, and/or a *Clerk* does exist whose key is *id*. For example, query q_c for *Clerk* would be:

```
SELECT * FROM R^PSEC_PE+ , R^C_C
WHERE R^PSEC_PE+.ssn=id AND p/e='E' AND R^PSEC_PE+.ssn=R^C_C.ssn
```

Assume the result of query q_c is not empty. In this case, the most specific class for the retrieved object will be *Clerk*; moreover, the result of query q_c will be used as values for the attributes of the retrieved object.

Update. An update can be the modification of either an attribute of an object or a link, described by some node. As it is customary in object-oriented programming, we assume that modifying the most specific class for an object is not allowed. In this case, the update of an object o is performed as stated in the standard semantics of M^2ORM^2, that is, by modifying tuples used to represent o.

Deletion. The deletion of an object o is performed by deleting tuples used to represent o. Again, for this case there is no difference with respect to the standard semantics of M^2ORM^2.

5 Correctness of mappings

Correctness is an important aspect of object/relational mappings. Intuitively, a mapping is correct if it supports, in an effective way, the management of CRUD operations on objects and links by means of the underlying relational database. We now briefly discuss correctness conditions concerning the mapping of inheritance hierarchies. For the sake of presentation, we now make the following assumptions: (i) each class is primary in exactly one node; (ii) each node contains at most one relation. (For a treatment of cases where the above assumptions do not hold we refer the reader to previous work [5, 6].) In this case, correctness of a mapping requires, at least, the following conditions to hold:

- for each concrete class C, each of the attributes of C and of its superclasses is mapped to exactly one relation attribute (apart from possible foreign keys), among the relations that store C;
- for each relation R and each class C stored by R, each of the attributes of R is mapped to at most one class attribute;
- key attributes of classes are related to key attributes of relations;

– class attributes that can be null are related to relation attributes that can be null.

The mapping described by Example 2 satisfies these conditions, and indeed it is a correct mapping between (the hierarchy of) the object schema of Fig. 1 and (part of) the relational schema of Fig. 2. We would like to point out that, using other M^2ORM^2 mapping features, it is possible to define a correct mapping between the whole schemas shown by Fig. 1 and 2.

6 Related work

There are several object/relational mapping tools available today (for a comparison of some tools supporting Java see *http://c2.com/cgi/wiki?ObjectRelationalToolComparison*); some of them also offer the meet-in-the-middle approach (e.g., [10, 15]). A mainstream application of ORM tools is supporting container-managed persistence (CMP) of Entity Beans in J2EE application servers [18]. Major relational DBMS vendors have recently started offering object/relational mapping tools based on the meet-in-the-middle approach, e.g., Oracle AS TopLink [16] and Microsoft ObjectSpaces [13].

In ORM tools, mappings are usually represented by graphs, as we do in M^2ORM^2. For example, in TopLink [16] a mapping comprises *descriptors* (corresponding to nodes) and *relationships* (corresponding to (relationship) arcs). Often, each node relates just a single class to just a single relation. Some systems (e.g., ObjectSpaces [13]) allow expressing more complex mappings between groups of classes and groups of relations, as we do in M^2ORM^2.

Most ORM tools take inheritance hierarchies into account. However, they do not always offer all the three basic representation strategies for inheritance hierarchies described in Section 2.3. Furthermore, they usually permit to select, for each hierarchy, a single representation strategy to be applied to the whole hierarchy. For example, ObjectSpaces [13] allows for all three strategies but, to the best of our understanding, they cannot be applied separately to different parts of a single hierarchy. In Hibernate [10], different strategies can be applied to distinct parts of a same hierarchy, but with some limitations: in particular, if C_1 and C_2 are two direct subclasses of a same class C, it is not possible to apply *SR* to C and C_1 and *CR* to C and C_2. On the other hand, M^2ORM^2+*HIE* offers more mapping possibilities with respect to object schemas containing hierarchies. Indeed, none of these systems allows expressing the mapping described by Example 2.

The "professional" literature is rich of works on several aspects concerning the implementation of ORM tools. Many contributions on the topic are now available as book chapters (e.g., [1, 9]) or as web resources (e.g., [14]). On the other hand, the scientific literature on ORM tools (e.g., [17]), apart from our previous work [5, 6], is more limited or, simply, outdated by current technology offerings. The notion of mapping used in this paper is inspired from one proposed in the context of model management [3].

7 Discussion

In this paper we have introduced M^2ORM^2+*HIE*, a mapping model for object/relational mapping tools. With respect to our previous work [5, 6], in this paper we have investigated the problem of managing inheritance hierarchies. It turns out that, as other mapping tools, M^2ORM^2+*HIE* is able to manage the most common representations for hierarchies. However, unlike other available systems and proposals, in M^2ORM^2+*HIE* it is also possible to represent more complex mappings involving hierarchies, e.g., where the three basic strategies are applied independently to different parts of a multi-level hierarchy.

There are a number of aspects related to ORM tools that we would like to investigate in the context provided by M^2ORM^2 and M^2ORM^2+*HIE*; these include: data types, multi-attribute keys, complex attributes, polymorphic associations, and cascade semantics. We believe that such features can be introduced in our mapping model in a graceful way.

References

1. S.W. Ambler. *Agile Database Techniques.* Wiley Publishing, 2003.
2. C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach.* Benjamin-Cummings, 1992.
3. P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for the management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
5. L. Cabibbo and R. Porcelli. M^2ORM^2: A Model for the Transparent Management of Relationally Persistent Objects. In *International Workshop on Database Programming Languages (DBPL)*, pages 166–178, 2003.
6. L. Cabibbo. Objects Meet Relations: On the Transparent Management of Persistent Objects. In *Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 429–445, 2004.
7. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
8. R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 2003.
9. M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.
10. Hibernate. *http://www.hibernate.org/.*
11. Java Data Objects. *http://www.jdocentral.com.*
12. W. Keller. Mapping Objects to Tables: A Pattern Language. In *European Pattern Languages of Programming Conference (EuroPLoP)*, 1997.
13. Microsoft ObjectSpaces. *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadonet/html/objectspaces.asp.*
14. Object Architects. Patterns for Object/Relational Mapping and Access Layers. *http://www.objectarchitects.de/ObjectArchitects/orpatterns/.*
15. ObJect relational Bridge. *http://db.apache.org/ojb/.*
16. Oracle AS TopLink. *http://otn.oracle.com/products/ias/toplink/.*
17. J.A. Orenstein. Supporting retrievals and updates in an object/relational mapping system. *IEEE Bull. on Data Engineering*, 20(1):50–54, 1999.
18. E. Roman. *Mastering Enterprise JavaBeans.* Wiley Publishing, 2002.